UNLIMITED

3509398

(2)

AD-A250 769

Report No. 92004

Report No. 92004

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

DTIC
ELECTE
MAY 1 4 1992
S D
D

IMPROVING THE TRANSLATION FROM DATA FLOW DIAGRAMS INTO Z BY INCORPORATING THE DATA DICTIONARY

Author: G P Randell

This document has been approved for public release and sale; its distribution is unlimited.

92-12649

2 5 11 12

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

January 1992

UNLIMITED

0121567

CONDITIONS OF RELEASE

309398

.....

DRIC U

COPYRIGHT (c)
1988
CONTROLLER
HMSO LONDON

.....

DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

DEFENCE RESEARCH AGENCY

RSRE Report 92004

Title: Improving the Translation from Data Flow Diagrams into Z by Incorporating the Data Dictionary

Author: G P Randell

Date: January 1992

Abstract

Earlier work developed formal translation rules for generating a specification in the formal language Z from a data flow diagram The Z specification produced lacked detail, especially on the types of the data flowing around the system. This report describes how to use information from a data dictionary to improve the Z specification Formal translation rules from a data dictionary to Z are presented

Accesion For
NTIS CRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Copyright
©
Controller HMSO, London
1992

INTENTIONALLY BLANK

Contents

INTENTIONALLY BLANK

1. Introduction

Data flow diagrams are a commonly used tool of systems' analysis, and are used by many of the current structured methods for system development. They are used to represent pictorially the data flows within a required system, showing how data enters and leaves the system, what changes the data, and where the data is stored. As such they are an important technique for understanding and communicating the functionality of the system.

Data flow diagrams contain four types of symbols (elements). These are:

1. External entity: a source or recipient of data outside the system, represented by a rectangle.
2. Process: an activity which transforms or manipulates data, represented by a circle.
3. Data store: a collection of any type of data in any form, represented by two parallel horizontal lines.
4. Data flow: showing a movement of data, represented by an arrow with the arrow head indicating the direction of flow and a label showing what data is involved.

Earlier work [1] developed formal translation rules for generating a specification in the formal language Z [2,3] from such diagrams. However, the specification generated was an outline only, and contained no information on the types of the data flowing round the system or on what data is actually kept in each data store.

The reason for this lack of detail is that the information from which the types of the data can be deduced does not appear on a data flow diagram. Data flows are only labelled with the name of the data, and do not say anything more about the composition of that data. Rather, an accompanying data dictionary contains details about the data and also details of what data is stored in the data stores.

The purpose of this report is to explain how the information in a data dictionary may be incorporated into the Z specification generated from the relevant data flow diagram. A fuller, more complete Z specification will then result.

The strategy is to provide formal translation rules from a data dictionary to Z. This is achieved in the same manner as that used for the original translation from data flow diagrams into Z in [1]. That is, the data dictionary is specified in Z, and an abstract syntax for the relevant parts of Z specified, again in Z. A function is then defined between the specification of data dictionaries to the abstract syntax of Z, which gives the meaning of the data dictionary in Z This meaning function gives the translation rules

The remainder of this report is structured as follows. Section 2 describes the particular form of data dictionary which has been used for this work, and gives some examples of data dictionary entries and the equivalent Z. Section 3 presents the specifications of the data dictionary, Z and the translation rules, and section 4 contains the conclusions. An updated version of the translation from data flow diagrams into Z, incorporating the information from the data dictionary, is given in the annex.

2. The Data Dictionary

The particular form of data representation chosen for this work is that defined in [4]. This notation is based on the three basic operations of sequence, selection and iteration. The notation is summarised in the following table:

Symbol	Read As
=	is composed of
+	together with
[.. | .. | ..]	select one of
{ ... }	iterations of

This particular notation has been chosen because it is well used, easy to understand and is well defined. The three basic operations are those used in structured programming and in methods like JSD (Jackson System Development).

To give an idea of how a data dictionary expressed in this notation looks, consider the following examples. The data dictionary contains an entry for all types of the data in the system, so that the type of every data flow is defined.

Example 1

A data type which represents all the days of the week will be represented as:

DaysOfTheWeek = [Monday | Tuesday | Wednesday | Thursday
| Friday | Saturday | Sunday]

So each element of the type is one of the days Monday to Sunday. This is a selection type. The selections do not have to be simple as in this example, but can be any type.

Example 2

An iterated type is one in which a component is repeated some number of times. For example, a bank statement is composed of a number of transactions, and may be represented as:

Statement = { Transaction }

Iterated types may be annotated with lower and upper ranges, to constrain the number of iterations. This has the form $_m\{ \ldots \}_n$ where m and n are both greater than or equal to zero and m is not greater than n. If no range is given it is assumed that the component may be repeated some arbitrarily large number of times. This ensures the iteration type is finite. The type between the braces may be as complicated as is desired, for example it could be a composite type, or a selection type.

Example 3

The third sort of type is a composition type. The bank transactions from the previous example, is one of these. It is a combination of, say, the date of transaction, a description of the transaction (for example, whether it is a standing order, or direct debit, or cash withdrawal, etc.), whether it is a debit or credit, and the amount of money involved, and may be represented as:

Transaction = Date + Description + DebitOrCredit + Amount

As before, the composite parts of one of these types may be complicated. The ordering of the components is not important.

Example 4

If we do not wish to give the exact details of a type, we can simply give a description of the type. The description is a simple English sentence, with an asterisk at each end to denote the beginning and end (rather like a comment in a programming language). So, for example, we may not wish to give any further details about the amount of money involved in a transaction, from the previous example. The entry for "Amount" in the data dictionary would then be:

Amount = * The amount of money debited (or credited) at each transaction *

Descriptions can be added to any data dictionary entry, to help the reader understand the purpose of the entry. And all data types must have an entry in the data dictionary, even if it is only a description.

There are also entries in the data dictionary for all the data stores which appear on a data flow diagram. The difference with a data store definition is that the key of the store, that is, that part of the data store which acts as an 'index' into the store, is highlighted (by underlining). Consider the following example.

Example 5

This example is of a data store which contains a bank's database. That is, each entry in the store consists of the bank account number, the name of the customer, and the amount in the account. This would be represented as:

Bank = { AccountNo + Name + Amount }

This is an iterated type because a bank holds many accounts. The account number is the key. The key does not, in general, have to be a single component but may be a compound key when a single component is not sufficient to uniquely identify the entry. For example, a telephone book has a key comprising the subscriber's name and address, as the name alone is not sufficient to find the 'phone number.

To motivate the translation, consider the Z equivalents of the above examples. The first was a selection type, and it's equivalent in Z will be a free type as follows:

$$DaysOfTheWeek ::= Monday \mid Tuesday \mid Wednesday \mid Thursday \mid Friday \mid Saturday \mid Sunday$$

The second example was an iterated type. The nearest equivalent in Z to this is a sequence, as follows:

$$Statement == seq\ Transaction$$

where Transaction must also be defined. If range constraints were used, then a predicate would be needed to say that all statements were of an appropriate length. For example, if the range was 3 to 20, so that every statement had to contain at least three entries but no more than 20, then the following constraint would be needed:

$$\forall\ s : Statement \cdot 3 \leq \#s \leq 20$$

The third example was a composite type. The nearest equivalent to this in Z is a schema, as follows:

Transaction
---
da : Date
desc : Description
d_or_c : DebitOrCredit
am : Amount

where Date, Description, DebitOrCredit and Amount must be defined. The identifiers da, desc, d_or_c and am are added to the Z description to make the schema complete.

The fourth example had no actual definition, just a description. The Z equivalent of this is the given set, as follows:

$$[\ Amount\ ]$$

A Z specification containing such a given set should also have an English explanation of what the given set represents, which will

probably be much the same as the description which appears in the data dictionary.

And the final example is of a data store, with a key. This is again represented in Z as a schema, but containing a function from the key to the rest of the data store entry, as follows:

```
Bank
accounts : AccountNo ⇸ AccountDetails
```

AccountNo is the key and must be defined elsewhere. AccountDetails is a schema representing the type of the rest of the data store entry, constructed in the same way as that for a composition type. Thus the schema AccountDetails will be of the form:

```
AccountDetails
no : Name
am : Amount
```

In the cases where the key is compound, that is where it has more than one component, a schema is also needed to represent it, constructed in the same way. The identifiers in the schemas and the additional schema names themselves must be added to produce a correct Z specification.

The formal description of the rules for translating a data dictionary into Z are given in the following section.

3. Translation into Z

3.1 The Specification of the Data Dictionary

A specification, in Z, of the data dictionary is needed to formalise the data dictionary definitions so that later a function can be defined mapping these definitions on to their Z equivalents.

We start by introducing a given set to represent the description of each definition in the data dictionary.

```
[ description ]
```

The description is textual, but we need not be concerned about its exact form.

```
[ definition ]
```

The actual definitions are recursive, so we first introduce them as a given set There are five sorts of definition: compositions, iterations, selections, data store definitions and null definitions. The null definition occurs when just a description is given

operation ::= composition | iteration | selection

The three basic operations are specified

comp_def
op : operation
components : $\mathbb{F}_1$ definition
---
op = composition
#components ≥ 2

The first kind of definition is a composition definition This has the appropriate operation and a set of other definitions, which are the components of the composite type. There must be at least two components for the definition to be sensible For example, consider the composition definition given above, namely

Transaction = Date + Description + DebitOrCredit + Amount

This definition has the set of components {Date, Description, DebitOrCredit, Amount}.

iter_def
op : operation
component : definition
lower, upper : ℕ
---
op = iteration
lower ≤ upper

An iteration only has one component. the type between the braces. This type may be a complicated definition. Lower and upper range constraints on iteration definitions are given, to restrict the number of iterations. For example, consider the iteration definition given above, namely:

Statement = { Transaction }

The component here is "Transaction", the lower limit is 0 (by default) and the upper limit is an arbitrarily large number (the exact number is left unspecified as it will depend on the implementation).

```
select_def
op : operation
components : F1 definition
-----------------------------
op = selection
#components ≥ 2
```

As for composition types, a selection type has the appropriate operation and a set of other definitions, which are the components of the selection type. Again there must be at least two components for the definition to be sensible. For example, consider the selection definition given above, namely.

DaysOfTheWeek = [Monday | Tuesday | Wednesday | Thursday
| Friday | Saturday | Sunday]

This has the set of components {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}.

```
ds_def
key, rest : F definition
-----------------------------
#key > 0
```

A data store definition has a non-empty key which identifies the rest of the data store contents. For example, consider the data store definition given above, namely:

Bank = { AccountNo + Name + Amount }

This has key {AccountNo} and the rest is the set {Name, Account}. A data store may have no other contents, but just have a key. In this case, the schema generated will contain just a set of the key elements rather than a function

```
definition ::= comp<comp_def> | iter<iter_def>
        | sel<select_def> | null<seq Char> | ds<ds_def>
```

These are all brought together to form the free type describing definitions. Null definitions just have identifiers (names)

We can now construct data dictionary entries

```
data_dict_entry
name : seq Char
def : definition
desc : description
---
def ∈ rng null ⇒ null⁻¹def = name
```

Each entry has a name, which is an identifier, a definition and a description. In the case of entries with null definitions, the definition simply uses the name of the entry.

```
data_dictionary : ℙ (seq Char ⇸ data_dict_entry)
---
∀dd:data_dictionary · ∀n:dom dd · (dd n).name = n
```

A data dictionary is simply defined as a map from the types names to their entries. Throughout this specification it has been assumed that the data dictionary is sensible, that is it contains only valid entries and does not have anything like

Bank = {Bank},

for example If the dictionary is sensible, a useful Z specification will be produced. On the other hand, if the dictionary contains some circular definitions or other incorrect ones, then a Z specification may be generated, but will contain some type errors which may be found using a Z type-checker such as [3].

And this completes the Z specification of the data dictionary.

3.2 The Specification of Z

In order to provide translation rules into Z using the strategy described in section 1 above, we also need a specification of the relevant parts of Z. This specification is an extended version of that given in [1]. The extension is to encompass Z free types which are needed for the translation of selection types in the data dictionary, to allow syntactic definitions which are needed for the translation of iteration types and to allow a particular sort of predicate.

```
identifier == seq Char
```

We start by introducing identifiers, which are just sequences of characters (strings). To specify schemas we need to specify names, signatures and predicates: the three components of every schema

```
schema_name == seq Char
```

The name of a schema is just a sequence of characters. The signature is more complicated. There are two sorts of elements in a signature: inclusions and declarations. Inclusions are names of other schemas to be included, and declarations are (identifier,type) pairs. Identifiers have already been defined. Types are more difficult. In fact, the definitions of types and signatures are mutually recursive, so a set of all possible types is first introduced as a given set, and then later constrained.

```
[ type ]

signature_element ::= inc«schema_name»
                    | dec«(identifier×type)»

signature == ℙ signature_element
```

A signature is a set of signature elements, each of which is an inclusion or declaration

Free types are composed of a set of branches.

```
branch ::= simple«type» | constructed«(identifier×type)»
```

Branches are either simple ones consisting of the name of a set, or more complicated ones consisting of a constructor function applied to a type. For example, the free type `signature_element` above has two complicated branches. The first comprises the constructor function `inc` applied to the set of schema names, and the second comprises the constructor function `dec` applied to the set of (identifier,type) pairs Note that the sets referred to are actually types and the names of the constructor functions are identifiers.

The definition of types can now be completed.

```
type ::= given«seq Char» | tuple«seq type»
       | powerset«type» | schema_type«signature»
       | freetype«ℙ₁ branch»
```

There are five sorts of type in Z: those arising from given sets; tuple types, arising from Cartesian products; powerset types; schema types and free types.

The third part of a schema is the predicate.

```
[ predicate ]

| empty : predicate
```

Nothing is said about the construction of predicates, just that they exist. The empty predicate is defined.

```
┌ schema ──────────┐
│ n : schema_name  │
│ sig : signature  │
│ pred : predicate │
└──────────────────┘
```

A schema has a name, a signature and a predicate.

Three extra Z elements are introduced. First, a free type is needed for those elements defined as a selection in the data dictionary. Secondly, a syntactic definition is needed for those elements defined as an iteration. Thirdly, a predicate is needed for constraining the size of sequences produced from iterations.

```
┌ free_type_def ──────┐
│ n : seq Char        │
│ branches : F1 branch│
└─────────────────────┘
```

Free type definitions have a name and a set of branches.

```
z_element ::= given_set⟨seq Char⟩
            | free_type⟨free_type_def⟩ | box⟨schema⟩
            | syn_def⟨seq Char×type⟩
            | bounds⟨seq Char×ℕ×ℕ⟩
```

The parts of Z needed are given sets, free types, schemas, a particular form of syntactic definition and a particular form of predicate. The syntactic definition will be a sequence definition, with a name and the type of the elements of the sequence. The function syn_def takes this name and type and produces a Z sequence. The predicate, constructed using the function bounds, will be a restriction on the size of the sequence corresponding to a particular iteration definition.

```
z_specification == F z_element
```

So a Z specification is just a set of these Z elements. At a more concrete level, a Z specification is actually a sequence of elements, to ensure that rules of declaration before use are followed. However, this detail is unnecessary for the purpose of this report. This completes the specification

3.3 The Translation Rules

Having specified the data dictionary and the necessary parts of Z, the next stage is to specify the meaning function which provides the translation. There are two stages in this process: first, a function which generates the appropriate Z type from each data dictionary entry must be defined; and secondly, the function to translate each data dictionary entry into the Z construct which will appear in the generated specification must then be defined.

```
translate_def : definition → type
```

The function to generate the Z types will be defined in parts, one part for each of the sorts of definition which may appear in a data dictionary, and it is recursive. Remember that each data dictionary entry has three parts: a name, a definition and a description; and that there are five sorts of definition: compositions, iterations, selections, data store definitions and null definitions.

```
get_ids : signature → ℙ identifier
─────────────────────────────────────────────
∀sig:signature
 • get_ids sig = {i:identifier; ty:type | dec(i,ty)∈sig • i}
```

```
type_of : identifier → type
```

```
in_branch : identifier ↔ type
```

Extra functions are needed which retrieve the identifiers used in the signature of a schema from that signature, associate identifiers with their types and relate identifiers and types which together make up a complicated branch of a free type. The second and third of these are left unspecified.

```
trans_pars
dd : definition
zt : type
```

```
trans_comp
trans_pars
---
dd ∈ rng comp
zt = schema_type (sig)
where
  sig : signature
  I : ℙ1 Identifier
  ---
  (sig,I) ∈ get_ids
  sig = {i:I;ty:type
        | ty ∈ translate_def ( (comp⁻¹ dd).components )
         ∧ type_of i = ty
        • dec (i,ty)}
```

Composition definitions will be translated into Z schemas, so the Z type they give rise to is a Z schema type The Z schema will have one component for each of the components of the composition, so the schema type has a signature containing one declaration for each of these. The actual identifiers which appear in the signature are left unspecified, and an anonymous set I is provided.

```
trans_iter
trans_pars
---
dd ∈ rng iter
zt=powerset(tuple<given ("Nat"),
                  translate_def (iter⁻¹ dd) component>)
```

Iteration definitions will be translated into sequences In Z, sequences are really functions from the natural numbers to the type of the elements of the sequence. So the Z type for an iteration is a powerset type of a tuple (sequences are just functions and functions are really just sets of pairs) The first part of the tuple type is the natural numbers (the domain of the sequence), and the second part is the type of the iterated component (the range of the sequence)

```
trans_sel
trans_pars
─────────────────────────────────────────
dd ∈ rng sel
zt = freetype (ft)
where
ft : ℙ₁ branch
op : ℙ₁ identifier
─────────────────────────────────────────
#ft = #op = #(sel⁻¹ dd).components
ft = {o:op;ty:type
      | ty ∈ translate_def ((sel⁻¹ dd).components)
      ∧ (o,ty) ∈ in_branch
      • constructed(o,ty)}
```

A selection definition will be translated into a Z free type. Thus the Z type generated is a free type, and it has the same number of branches as there are components in the selection. Each branch is constructed from an operation (the name of the constructor function), and the type of one of the selection components. All the branches generated are complicated branches. If, by later examination, it was realised that the type of a selection component contained only one element, then the complicated branch generated for that element could be replaced by a simple branch (the constructor function would, in effect, be the identity function).

```
make_def : ℙ₁ definition ⇸ definition
─────────────────────────────────────────
∀ defs : ℙ₁ definition; def : definition
• def = make_def defs ⇔
   #defs = 1 ∧ {def} = defs
  ∨ #defs > 1 ∧ def = comp cd
where
cd : comp_def
─────────────
cd.components = defs
```

A preliminary function is introduced before we specify the translation of a data store. This function takes a set of definitions and, if the set has more than one element, it generates a new composition definition with components being those in the original set. The purpose of this is to allow the key and the rest of the elements of a data store to be treated exactly the same as composition definitions for the purpose of

generating Z schemas and types from them. For example, in the example given in section 2 above of a data store, the components other than the key (Name and Amount) will be grouped together into one composition definition, and the rules for translating composition definitions into Z applied to generate the schema Account Details.

```
trans_ds
trans_pars
─────────────────────────────────────────────
dd ∈ rng ds
zt = schema_type (sig)
where
  sig : signature
  i : identifier
  pt : type
─────────────────────────────────────────────
  sig = ( dec (i, powerset pt) )
  #((ds⁻¹ dd).rest) = 0
       ∧ pt = translate_def(make_def((ds⁻¹ dd).key))
  ∨
  #((ds⁻¹ dd).rest) > 0
   ∧ pt = tuple
            ⟨ translate_def(make_def((ds⁻¹ dd).key)),
              translate_def(make_def((ds⁻¹ dd).rest)) ⟩
```

Data store definitions will also be translated into Z schemas, so the Z type produced is a schema type. As in the final example in section 2 above, the signature of the schema contains just a function, mapping the key of the data store to the rest of the components, unless there are no other components in which case the signature contains just a set of the key elements.

```
trans_null
trans_pars
──────────────────
dd ∈ rng null
zt = given (null⁻¹ dd)
```

The final definition is the null definition This gives rise to a Z given set, with name the same as the definition

$$\forall trans\_pars \cdot zt = translate\_def\ dd \Leftrightarrow trans\_comp \vee trans\_iter \vee trans\_sel \vee trans\_ds \vee trans\_null$$

All these parts are brought together to define the function.

We can now define the function which translates each data dictionary entry into its equivalent Z construct(s).

$$translate\_entry : data\_dict\_entry \rightarrow \mathbb{F}\ z\_element$$

As for the previous case this translation function will be defined in parts.

trans_entry_pars

$$dde : data\_dict\_entry$$
$$ze : \mathbb{F}\ z\_element$$

trans_entry_comp

$$trans\_entry\_pars$$

$$dde.def \in rng\ comp$$
$$ze = \{ b \cdot s \}$$

where

$$s : schema$$

$$s.n = dde.name$$
$$s.pred = empty$$
$$s.sig = schema\_type^{\prime}\ (translate\_def\ dde.def)$$

The Z schema generated for a composition definition uses the name of the data dictionary entry as its name, and has an empty predicate. Any predicates constraining the values of the entry could be added later if desired.

```
trans_entry_iter
trans_entry_pars
─────────────────────────────────────────
dde.def ∈ rng iter
ze = (syn_def (dde.name,
        translate_def (iter⁻¹ dde.def).component),
      bounds (dde.name, (iter⁻¹ dde.def).lower,
            (iter⁻¹ dde.def).upper))
```

An iteration gives rise to two Z constructs. One is the syntactic definition which defines the sequence of the type of the iterated component, and the other is a predicate which constrains the size of the sequence.

```
trans_entry_sel
trans_entry_pars
─────────────────────────────────────────
dde.def ∈ rng sel
ze = { free_type ftd }
where
  ftd : free_type_def
  ──────────────────────
  ftd.n = dde.name
  ftd.branches = freetype⁻¹ (translate_def dde.def)
```

Selections give rise to a Z free type. Most of the work in generating this free type has been done by the `translate_def` function, which generated all the branches of the free type. All that is done here is to give the free type an appropriate name.

Data stores potentially give rise to three schemas, one for the data store itself, one for the key, and one for the rest of the components of the store. Schemas are only generated for the last two of these if they contain more than one element.

```
trans_entry_ds_1
dde : data_dict_entry; others : ℙ z_element
─────────────────────────────────────────
#(ds⁻¹ dde.def).key = 1 ∧ #(ds⁻¹ dde.def).rest ≤ 1
others = {}
```

If both the key contains only one element and the rest of the components contains at most one, then no extra Z element is generated.

```
trans_entry_ds_2
dde : data_dict_entry
others : ℙ z_element
─────────────────────────────────────────
#(ds⁻¹ dde.def).key = 1 ∧ #(ds⁻¹ dde.def).rest > 1
others = { box (t) }
where
  t : schema; tn : identifier
  ─────────────────────────────────
  t.n = tn ∧ type_of tn = schema_type t.sig
  t.pred = empty
  t.sig = schema_type⁻¹
      translate_def (make_def (ds⁻¹ dde.def).rest)
```

If the key has only one element but the rest of the components has more than one, then one schema is generated to represent the type of the rest of the components. This schema has a suitable name, an empty predicate, and a signature constructed by translating the composition definition made from the rest of the components into a Z schema type.

```
trans_entry_ds_3
dde : data_dict_entry
others : ℙ z_element
─────────────────────────────────────────
#(ds⁻¹ dde def).key > 1 ∧ #(ds⁻¹ dde.def).rest ≤ 1
others = { box (t) }
where
  t : schema; tn : identifier
  ─────────────────────────────────
  t.n = tn ∧ type_of tn = schema_type t.sig
  t.pred = empty
  t.sig = schema_type⁻¹
      translate_def (make_def (ds⁻¹ dde.def).key)
```

Similarly, if the key has more that one element but the rest of the components has either none or one, then one schema is generated to represent the type of the key. Again this schema has a suitable name, an empty predicate, and a signature constructed by translating the composition definition made from the key into a Z schema type.

```
trans_entry_ds_4
dde : data_dict_entry
others : ℙ z_element
─────────────────────────────────────────────
#(ds⁻¹ dde.def).key > 1 ∧ #(ds⁻¹ dde.def).rest > 1
others = { box (t), box (u) }
where
  t,u : schema; tn, un : identifier
  ─────────────────────────────────────────
  t.n = tn ∧ type_of tn = schema_type t.sig
  t.pred = empty
  t.sig = schema_type⁻¹
      translate_def (make_def (ds⁻¹ dde.def).key)
  u.n = un ∧ type_of un = schema_type u.sig
  u.pred = empty
  u.sig = schema_type⁻¹
      translate_def (make_def (ds⁻¹ dde.def).rest)
```

If both the key and the rest of the components have more than one elements, then two schemas are generated, one to represent the type of the key and the other to represent the type of the rest of the components. These schemas are constructed in the same way as before.

```
trans_entry_ds
trans_entry_pars
─────────────────────────────────────────────
dde.def ∈ rng ds ∧ ze = { box s } ∪ others
where
s : schema; others : ℙ z_element
─────────────────────────────────────────────
s.n = dde.name ∧ s.pred = empty
s.sig = schema_type⁻¹ (translate_def dde.def)
trans_entry_ds_1 ∨ trans_entry_ds_2 ∨ trans_entry_ds_3
 ∨ trans_entry_ds_4
```

Finally, the schema representing the data store itself is generated, with the same name as the data dictionary entry, and with an empty predicate. The other Z schemas generated, where needed, for the key and the rest of the components of the data store, are combined with the schema representing the data store to give the total set of Z elements produced from a data store definition

```
trans_entry_null
trans_entry_pars
─────────────────────────
dde.def ∈ rng null
ze = { given_set dde name }
```

The last part of the function generates a Z given set for all those data dictionary entries which just have descriptions.

```
∀trans_entry_pars • ze = translate_entry dde ⇔
 trans_entry_comp ∨ trans_entry_iter ∨ trans_entry_sel
 ∨ trans_entry_ds ∨ trans_entry_null
```

All these parts are brought together to define the function

```
translate_d_d : data_dictionary → z_specification
─────────────────────────────────────────────
∀dd data_dictionary; zs z_specification •
   zs = ⋃ translate_entry (| rng dd |)
```

We can now translate the whole data dictionary into a Z specification by simply applying the translate entry function to each entry in the data dictionary.

4. Conclusions

The Z specification generated from a data flow diagram using the rules presented in [1] is an outline only. In particular, no information about the type (composition) of the data flowing around the system is present. The formal translation presented in this report fills that gap by using the data dictionary which accompanies each data flow diagram to generate type information automatically. The Z specification generated from the diagram and dictionary together is a fuller, more useful specification.

In addition, a data flow diagram says nothing about the content of each data store on the diagram, whereas the data dictionary contains an entry for each describing the data held. Thus the Z schema generated from the diagram for each data store, which is in effect an empty schema box, should be replaced by the schema generated from the data dictionary description. Again, this will lead to a more useful Z specification being produced.

The annex to this report combines both translations into one formal translation from the diagram and dictionary together into a single Z specification.

Thus translations have been developed which formalise both data flow diagrams and their accompanying data dictionary, and which enable a useful Z specification to be generated automatically from them. However, the Z specification does not capture the communication aspects of the data flow diagram very well, so is still limited, especially when the data flow diagram contains mainly communications between processes directly and not via data stores In order to overcome this deficiency and enable communication aspects to be reasoned about, a translation from data flow diagrams into Hoare's CSP (Communicating Sequential Processes) has also been developed, and is presented in [5]

References

[1] G P Randell, Translating Data Flow Diagrams into Z (and vice versa), RSRE Report 90019, October 1990

[2] C T Sennett, Review of Type Checking and Scope Rules of the Specification Language Z, RSRE Report 87017, 1987

[3] G P Randell, ZADOK User Guide, RSRE Memorandum 4356, 1990

[4] P T Ward & S J Mellor, Structured Development for Real-Time Systems, Volume 1: Introduction and Tools, Yourdon Press, 1985

[5] G P Randell, Data Flow Diagrams and CSP, RSRE Report (in preparation), 1992

INTENTIONALLY BLANK

Annex - Translating Data Flow Diagrams into Z

A.1 Introduction

The purpose of this annex is to bring together an updated version of the translation from data flow diagrams into Z originally presented in [1] and the translation presented in the main part of this report to automatically generate a more useful Z specification. The same strategy is adopted, in that first a specification of data flow diagrams is given, then a specification of the relevant parts of Z, and finally the translation (meaning) function.

The specification of Z needed for this translation is the same as that given in section 3.2 of the main part of this report, and will not be repeated in this annex. The translation rules presented make use of the data dictionary to Z translation rules from section 3.3.

A.2 The Specification of Data Flow Diagrams

The interpretation put on a data flow diagram by this formalization is that it represents operations being carried out on a state. The state is represented by the data stores, and the operations by the processes. The external entities just provide or receive data.

The different elements of a data flow diagram, namely external entities, processes and data stores, are specified first. The data flows between them are then specified as constraints on the set of diagram elements that make up a given diagram.

```
external_entity
names : IF identifier
```

The relevant part of an external entity, from the point of view of this translation, is the data which it sends to and receives from the system. The data moving around the system is shown by the names on the data flows. These names are modelled as simple identifiers.

Each process on a data flow diagram inputs data, transforms or manipulates it, and produces outputs.

```
process
inputs, outputs : IF identifier
reads, writes : IF identifier
process_name : seq Char
```

Inputs and outputs come from or go to other processes or external entities Communications with data stores are treated as reads or writes of the state data held in the store (a flow from a process to a data store being a write, and the reverse direction a read). The inputs, outputs, reads and writes correspond to the names on the appropriate data flows. Processes also have names, which are intended to indicate the function that the process performs.

Data stores represent the state of the system, upon which processes act by reading from or writing to them.

```
┌─ datastore ──────────────────────┐
│ reads, writes : F identifier     │
│ store_name : seq Char            │
└──────────────────────────────────┘
```

Reads from and writes to a data store correspond to the names on the flows to and from processes, respectively. Data stores also have names. The contents of a data store do not appear on a data flow diagram, so no mention of contents appears in this specification.

```
DFD_element ::= ext ⟨ external_entity ⟩
              | proc ⟨ process ⟩ | store ⟨ datastore ⟩
```

The three schemas external_entity, process and datastore represent the three possible types of data flow diagram element. In addition to diagram elements, a data flow diagram also contains data flows

```
┌─ data_flow ──────────────────────────────┐
│ origin, dest : DFD_element               │
│ label : identifier                       │
├──────────────────────────────────────────┤
│ origin ≠ dest                            │
│ origin ∈ rng proc ∨ dest ∈ rng proc      │
└──────────────────────────────────────────┘
```

Each data flow has an origin and a destination, both of which are diagram elements. Each is also labelled with the name of the data which flows along it. A data flow cannot be circular, in that it cannot return to its own origin, and it must have a process as either its origin or destination (or both). This prevents external entities having direct access to state data (except via processes to read or write that data), and also prevents a flow between two data stores, as data stores are passive.

A well-formed data flow diagram consists of a collection of diagram elements, connected by suitable data flows. The first check is on the connectedness of the diagram

```
Connected
elements : ℙ DFD_element
flows : ℙ data_flows
---
∀e:elements • (∃f:flows • e = f.origin ∨ e = f.dest)
∀f:flows • (f.origin ∈ elements ∧ f.dest ∈ elements)
```

The first predicate says that, for each element (external entity, process or data store), there must be a data flow either from it or to it (or both). This is to ensure that no element is "orphaned". The second says that all flows must be connected to elements in the diagram. Further constraints could be added, if required, to ensure that there were no sources or sinks of data on the diagram, or that the diagram was completely connected in that it could not be split into two groups of elements, with no data flows between any two elements from the different groups.

There are also checks on a data flow diagram to ensure that the representation in terms of elements and data flows, as in the Z specification above, is consistent with the diagram. The first check is on external entities.

```
External_entities_ok
elements : ℙ DFD_element
flows : ℙ data_flows
---
∀e:elements; ex:external_entity | e = ext ex •
  ex names = {f:flows | e=f.origin ∨ e=f.dest • f.label}
```

This check ensures that the names recorded for each entity correspond to the names on the data flows to and from that entity

The second check is on processes

```
Processes_ok
elements : ℙ DFD_element
flows : ℙ data_flows
---
∀e:elements; p:process | e = proc p ·
p.inputs = {f:flows | f.origin ∈ (rng ext ∪ rng proc) ∧
            f.dest = e · f.label}
p.outputs = {f:flows | f.origin=e ∧ f.dest ∈ (rng ext ∪
             rng proc) · f.label}
p.reads = {f:flows | f.origin ∈ rng store ∧ f.dest=e
          · f.label}
p.writes = {f:flows | f.origin=e ∧ f.dest ∈ rng store
          · f.label}
```

This schema says four things First, that the inputs to a process correspond to the names on the data flows to that process from either external entities or other processes. Secondly, that the outputs from a process correspond to the names on the data flows from that process to either external entities or other processes. Thirdly, that the "reads" a process performs corresponds to the names on the data flows from data stores to that process. And, fourthly, that the "writes" a process performs correspond to the names on the data flows from that process to data stores

The third check is on data stores

```
Datastores_ok
elements : ℙ DFD_element
flows : ℙ data_flows
---
∀e:elements; d:datastore | e = store d ·
 d.writes = {f:flows | f origin ∈ rng proc ∧ f.dest=e
          · f.label}
 d.reads = {f:flows | f.origin=e ∧ f.dest ∈ rng proc
          · f.label}
```

This check ensures that the data written to a data store corresponds to the names on the data flows from processes to that data store, and that the data read from a data store corresponds to the names of the data flows to processes from that data store

```
DFD
elements : F DFD_element
flows : F data_flows
---
Connected ∧ External_entities_ok ∧ Processes_ok ∧
  Datastores_ok
```

A valid data flow diagram passes all the checks.

A.3 The Specification of the Translation Rules

The translation takes a data flow diagram and its accompanying data dictionary and produces a Z specification. The types of each data item are found from the data dictionary, as are the data store schemas. The diagram gives rise to one operation schema for each process.

We first need to relate data flow diagrams to data dictionaries.

```
appropriate : DFD ↔ data_dictionary
---
∀dfd:DFD;dd:data_dictionary • appropriate(dfd,dd) ⇔
{ds:dfd.elements | ds ∈ rng store • (store~ ds).store_name}
  ∪ {df:dfd.flows • df.label} ⊆ dom dd
```

A data flow diagram and a data dictionary are related if the data dictionary contains an entry for every data flow (the labels on data flows are the names of the data types) and every data store on the diagram. The data dictionary must also contain entries for all definitions which appear as components of an entry. Thus, for example, if a data flow label "Address" had the following entry:

Address = HouseName + Number + Street + Town + County

then the data dictionary must also contain individual entries for "HouseName", "Number", "Street", "Town" and "County". This 'completeness' constraint is left unspecified. It should be noted that if the data dictionary is not complete then the Z specification generated will simply not type-check

The translation function for the diagram takes each process from a particular diagram with its associated data dictionary and produces an appropriate Z schema which describes the effect the operation represented by the process has on the state of the system. That is, it describes which data stores are affected The data dictionary is used to look up the label on each data flow to find the type of the data flowing

The signature of the schema generated from a process consists of just those data stores which are read by the process but not altered in any way, those data stores which the process does affect, and the inputs and outputs.

```
generate_reads : (process × DFD × data_dictionary)
                 ⇸ signature
─────────────────────────────────────────────────
∀p:process;dfd:DFD;dd:data_dictionary
 | proc p ∈ dfd.elements ∧ appropriate(dfd,dd)
• generate_reads (p,dfd,dd) =
  { df:dfd.flows;ds:datastore | store ds ∈ dfd.elements
  ∧ df.dest=proc p ∧ df.origin=store ds ∧
  ¬(∃df':dfd.flows • df'.origin=proc p∧df'.dest=store ds)
   • inc ("Ξ" ⌢ ds.store_name) }
```

Data stores which are read by the process but not altered in any way are prefixed by "Ξ" in the signature of the schema produced. These are found by looking for data stores in the diagram from which there is a data flow to the process in question, but for which there is no data flow in the opposite direction. Each data store which is the destination of a data flow is changed.

```
generate_writes : (process × DFD × data_dictionary)
                  ⇸ signature
─────────────────────────────────────────────────
∀p:process;dfd:DFD;dd:data_dictionary
 | proc p ∈ dfd.elements ∧ appropriate(dfd,dd)
• generate_writes (p,dfd,dd) =
  { df:dfd.flows; ds:datastore
  | df.dest=store ds ∧ df.origin=proc p ∧
    store ds ∈ dfd.elements
  • inc ("Δ" ⌢ ds.store_name) }
```

Those data stores which the process does affect are prefixed by "Δ" in the signature of the generated schema

```
generate_ins : (process x DFD x data_dictionary)
                    ⇸ signature
```

```
∀p:process;dfd:DFD;dd:data_dictionary
 | proc p ∈ dfd.elements ∧ appropriate(dfd,dd)
• generate_ins (p,dfd,dd) =
  {i:p.inputs;ip:identifier
   |last ip='?'∧ type_of ip = translate_def((dd i).def)
   • dec (ip,translate_def (dd i).def)}
```

Inputs are given an identifier ending with "?" in the signature in the usual Z style The types of these are found by looking up the relevant entry in the appropriate data dictionary.

```
generate_outs : (process x DFD x data_dictionary)
                    ⇸ signature
```

```
∀p:process,dfd DFD;dd:data_dictionary
 | proc p ∈ dfd elements ∧ appropriate(dfd,dd)
• generate_outs (p,dfd,dd) =
  {o:p outputs;op:identifier
   |last op='!'∧ type_of op = translate_def((dd o) def)
   • dec (op,translate_def (dd o) def)}
```

Similarly, outputs are given an identifier ending with '

```
trans_proc (processxDFDxdata_dictionary) ⇸ z_element
```

```
∀p:process,dfd DFD,dd data_dictionary,ze z_element
 | proc p ∈ dfd elements ∧ appropriate(dfd,dd)
• trans_proc (p,dfd,dd) = ze ⇔ ze = box s
where
 s : schema
```

```
 s n = p.process_name ∧ s pred = empty
 s sig = generate_reads (p,dfd,dd)
       ∪ generate_writes (p,dfd,dd)
       ∪ generate_ins (p,dfd,dd)
       ∪ generate_outs (p,dfd,dd)
 #s sig = #(p inputs ∪ p outputs ∪ p reads ∪ p writes)
```

So the schema generated is given the same name as the process and the empty predicate as information relating to the predicate part of the schema does not appear on a data flow diagram nor in the data dictionary. The four functions defined above are used to generate the signature of the schema, which has one element for each read, write, input and output.

```
translate : (DFD × data_dictionary) ⇸ z_specification
```

```
∀ dfd:DFD; dd:data_dictionary | appropriate(dfd,dd)
• translate (dfd,dd) = {p:process | proc p ∈ dfd.elements
    • trans_proc (p,dfd,dd)} ∪ translate_d_d dd
```

To generate a Z specification from a data flow diagram and a data dictionary together we simply translate all the processes on the diagram into Z schemas and and put these together with all the type definitions and data store definitions generated from the data dictionary Note that the diagram only provides the processes, that is the operations in the Z specification, and that the data dictionary must be appropriate for the diagram Thus the translations into Z from the data flow diagram and the data dictionary cannot be inconsistent.

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification, eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
REPORT 92004	JANUARY	1992

Originators Name and Location
RSRE, ST ANDREWS ROAD
MALVERN, WORCS WR14 3PS

Monitoring Agency Name and Location

Title
IMPROVING THE TRANSLATION FROM DATA FLOW DIAGRAMS INTO Z BY INCORPORATING THE DATA DICTIONARY

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
RANDELL, G P	29

Abstract

Earlier work developed formal translation rules for generating a specification in the formal language Z from a data flow diagram. The Z specification produced lacked detail, especially on the types of the data flowing around the system. This report describes how to use information from a data dictionary to improve the Z specification. Formal translation rules from a data dictionary to Z are presented.

Abstract Classification (U, R, C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

INTENTIONALLY BLANK